



Course Code : CST207
Course Name : Design and Analysis of Algorithms
Lecturer : Dr. Mohammed N. M. Ali
Academic Session : 2025/09
Assessment Title : AI-Driven Sorting Algorithm Optimizer
Submission Due Date : 28th December 2025

Prepared by :	Student ID	Student Name
	DSC2409006	Cui Zeyu
	DSC2409010	Fan Li
	DSC2409012	He Bingguang
	DSC2409042	Yu Fangrui
	DSC2409043	Yu Mingquan

Date Received : _____

Feedback from Lecturer:	
	Mark:

Own Work Declaration

I/We hereby understand my/our work would be checked for plagiarism or other misconduct, and the softcopy would be saved for future comparison(s).

I/We hereby confirm that all the references or sources of citations have been correctly listed or presented and I/we clearly understand the serious consequence caused by any intentional or unintentional misconduct.

This work is not made on any work of other students (past or present), and it has not been submitted to any other courses or institutions before.

Signature:

崔泽禹 范仟 何秉光 刘芳瑞 余铭泉

Date: 21st December 2025

AI-Driven Sorting Algorithm Optimizer

Report

CST207 Design and Analysis of Algorithms Group Project

Academic Session: 2025/09

Oral Presentation Video Link

https://drive.google.com/drive/folders/1YNGKs7LQVs_FJW2VaJKDxGBAhHr5GaCm

Table of Contents

1	Introduction	1
2	System Overview	1
3	Algorithm Design	2
3.1	Dataset Generation	2
3.2	Sorting Algorithms	3
3.3	AI Module: Decision Tree Approach	5
4	Implementation Details	7
4.1	Performance Measurement	7
4.2	Safety and Optimization	8
4.3	Multiple Versions of Optimizer	8
4.4	Compilation and Execution with Screenshots	10
5	Results and Analysis	12
5.1	Small Random Dataset (Size = 40)	12
5.2	Large Random Dataset (Size = 10,000)	13
5.3	Nearly Sorted Dataset (Size = 1,000)	14
6	Conclusion	15
A	CST207_Project_Group_202509_CLI.cpp	16
B	CST207_Project_Group_202509_GUI.cpp	25

Group Contribution Table

Student ID	Name	Specific Contribution Description
DSC2409006	Cui Zeyu	Implement Result Visualization GUI with Qt, Write Report and Edit Oral Presentation Video
DSC2409010	Fan Li	Implement AI Model with kNN and Decision Tree
DSC2409012	He Bingguang	Implement AI Model with kNN and Decision Tree
DSC2409042	Yu Fangrui	Implement Dataset Generation for All Types and Testing
DSC2409043	Yu Mingquan	Implement Sorting Algorithms and Measure Performance

1 Introduction

In modern software engineering, sorting is a fundamental operation whose performance critically depends on the characteristics of the dataset—such as size, order (sortedness), and element uniqueness. While standard libraries often use a one-size-fits-all approach (typically Quicksort or Introsort), no single algorithm is optimal for every scenario.

The goal of this project is to design and implement an **AI-Driven Sorting Algorithm Optimizer**. This system acts as an intelligent library that analyzes the input dataset's features and automatically selects the most efficient sorting algorithm. By integrating Artificial Intelligence (Decision Tree model) with traditional algorithmic analysis, we demonstrate how heuristics can optimize computational resource usage.

2 System Overview

In Figure 1, we illustrate the overall architecture of the system:

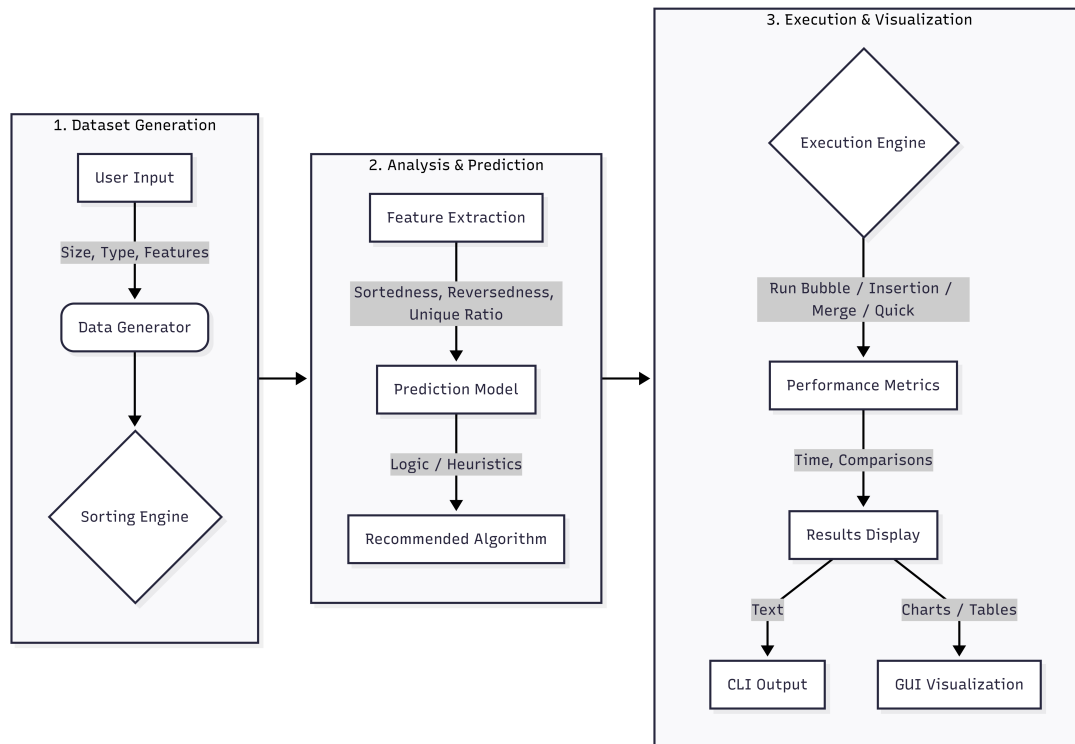


Figure 1: System Architecture Overview

The system consists of three main components:

1. **Dataset Generation:** Generate five dataset types (random, nearly sorted, reversed, few-unique, large random) with configurable size and distribution for comprehensive benchmarking. The generated datasets will be sorted by algorithms.
2. **Analysis and AI Module Prediction:** Extract features (size, sortedness, reversedness, unique ratio) and use decision-tree models to recommend the most suitable sorting algorithm. The decision tree in the AI module is constructed based on empirical observations and theoretical analysis of algorithm performance.
3. **Execution and Visualization:** Execute chosen algorithms, measure runtime and comparisons, and present results via CLI text and GUI tables. AI predictions are compared against actual performance to validate accuracy and shown visually.

3 Algorithm Design

Note that: To make the layout neater, all code in the main body of the report is un-commented. The relevant explanations will be provided in the text. Additionally, well-commented code will be added in full to the appendix.

3.1 Dataset Generation

To robustly test the algorithms, we implemented a generator capable of creating five distinct types of datasets, as required by the project specifications:

1. **Random Dataset:** Elements are generated using a random number generator with values in range $[1, size \times 10]$.

```
1 static vector<int> generateRandomDataset(int size) {  
2     vector<int> arr(size);  
3     srand(time(nullptr));  
4     for (int i = 0; i < size; i++) {  
5         arr[i] = 1 + rand() % (size * 10);  
6     }  
7     return arr;  
8 }
```

2. **Nearly Sorted Dataset:** A sorted array is created, and then approximately 10% of the elements are randomly swapped to simulate slightly disordered data.

```
1 static vector<int> generateNearlySorted(int size) {  
2     vector<int> arr(size);  
3     for (int i = 0; i < size; i++) arr[i] = i + 1;  
4  
5     int swaps = size / 10;  
6     srand(time(nullptr));  
7     for (int i = 0; i < swaps; i++) {  
8         int idx1 = rand() % size;  
9         int idx2 = rand() % size;  
10        swap(arr[idx1], arr[idx2]);  
11    }  
12    return arr;  
13 }
```

3. **Reversed Dataset:** Elements in the dataset are strictly arranged in descending order ($N, N - 1, \dots, 1$).

```
1 static vector<int> generateReversed(int size) {
2     vector<int> arr(size);
3     for (int i = 0; i < size; i++) {
4         arr[i] = size - i;
5     }
6     return arr;
7 }
```

4. **Few Unique Dataset:** The dataset contains many duplicate values, generated from a small pool of unique integers. First, a pool of `uniqueCount` random values is created, then the dataset is filled by randomly selecting from this pool.

```
1 static vector<int> generateFewUnique(int size, int uniqueCount) {
2     vector<int> uniqueValues;
3     srand(time(nullptr));
4     for (int i = 0; i < uniqueCount; i++) {
5         uniqueValues.push_back(rand() % 100 + 1);
6     }
7
8     vector<int> arr(size);
9     for (int i = 0; i < size; i++) {
10        arr[i] = uniqueValues[rand() % uniqueCount];
11    }
12    return arr;
13 }
```

5. **Large Random Dataset:** A stress-test dataset with size $N \geq 10,000$ to evaluate asymptotic performance. Uses the same `generateRandomDataset()` method but enforces a minimum size constraint of 10,000 elements in the main program.

3.2 Sorting Algorithms

We implemented four classical sorting algorithms:

- **Bubble Sort ($O(N^2)$):** Included for educational comparison; effective only on very small or nearly sorted data. Features early termination optimization: if no swaps occur in a pass, the array is already sorted.

```
1 static void bubbleSort(vector<int>& arr, long long& comparisons) {
2     int n = arr.size();
3     for (int i = 0; i < n - 1; i++) {
4         bool swapped = false;
5         for (int j = 0; j < n - i - 1; j++) {
6             comparisons++;
7             if (arr[j] > arr[j + 1]) {
8                 swap(arr[j], arr[j + 1]);
9                 swapped = true;
10            }
11        }
12        if (!swapped) break;
13    }
14 }
```

- **Insertion Sort ($O(N^2)$):** Highly efficient for small N or nearly sorted data due to low constant factors. Each element is inserted into its correct position in the sorted prefix. For nearly sorted data, the inner while loop terminates quickly, achieving near $O(N)$ performance.

```

1 static void insertionSort(vector<int>& arr, long long& comparisons) {
2     int n = arr.size();
3     for (int i = 1; i < n; i++) {
4         int key = arr[i];
5         int j = i - 1;
6         while (j >= 0) {
7             comparisons++;
8             if (arr[j] <= key) break;
9             arr[j + 1] = arr[j];
10            j--;
11        }
12        arr[j + 1] = key;
13    }
14 }

```

- **Merge Sort ($O(N \log N)$):** A stable, divide-and-conquer algorithm that guarantees performance but requires $O(N)$ auxiliary space. The algorithm recursively divides the array, then merges sorted subarrays.

```

1 static void mergeSort(vector<int>& arr, int l, int r, long long&
2     comparisons) {
3     if (l >= r) return;
4     int m = l + (r - l) / 2;
5     mergeSort(arr, l, m, comparisons);
6     mergeSort(arr, m + 1, r, comparisons);
7     merge(arr, l, m, r, comparisons);
8 }
9 static void merge(vector<int>& arr, int l, int m, int r, long long&
10     comparisons) {
11     int n1 = m - l + 1;
12     int n2 = r - m;
13     vector<int> left(n1), right(n2);
14
15     for (int i = 0; i < n1; i++) left[i] = arr[l + i];
16     for (int j = 0; j < n2; j++) right[j] = arr[m + 1 + j];
17
18     int i = 0, j = 0, k = l;
19     while (i < n1 && j < n2) {
20         comparisons++;
21         if (left[i] <= right[j]) {
22             arr[k++] = left[i++];
23         } else {
24             arr[k++] = right[j++];
25         }
26     }
27     while (i < n1) arr[k++] = left[i++];
28     while (j < n2) arr[k++] = right[j++];
29 }

```

- **Quick Sort ($O(N \log N)$):** *Optimization:* To prevent the worst-case $O(N^2)$ time complexity on *Reversed Datasets*, we implemented a **Randomized Pivot** strategy.

The pivot is selected randomly, ensuring $O(N \log N)$ performance on average for any input distribution.

```

1 static void quickSort(vector<int>& arr, int low, int high, long long&
  comparisons) {
2     if (low < high) {
3         int pi = partition(arr, low, high, comparisons);
4         quickSort(arr, low, pi - 1, comparisons);
5         quickSort(arr, pi + 1, high, comparisons);
6     }
7 }
8
9 static int partition(vector<int>& arr, int low, int high, long long&
  comparisons) {
10     int randomIndex = low + rand() % (high - low + 1);
11     swap(arr[randomIndex], arr[high]);
12
13     int pivot = arr[high];
14     int i = low - 1;
15     for (int j = low; j < high; j++) {
16         comparisons++;
17         if (arr[j] < pivot) {
18             i++;
19             swap(arr[i], arr[j]);
20         }
21     }
22     swap(arr[i + 1], arr[high]);
23     return i + 1;
24 }

```

3.3 AI Module: Decision Tree Approach

We implemented a Decision Tree to predict the best algorithm. The AI module extracts four key features from the dataset: *Size*, *Sortedness* (ratio of ascending pairs), *Reversedness* (ratio of descending pairs), and *Unique Ratio*. The feature extraction is implemented by analyzing consecutive pairs and using `unordered_set` to count unique elements:

```

1 static DatasetFeatures analyzeDataset(const vector<int>& data) {
2     DatasetFeatures features;
3     features.size = data.size();
4     features.isLargeDataset = (features.size > 1000);
5
6     long long ascendingPairs = 0, descendingPairs = 0;
7     for (int i = 0; i < features.size - 1; i++) {
8         if (data[i] <= data[i+1]) ascendingPairs++;
9         if (data[i] >= data[i+1]) descendingPairs++;
10    }
11
12    features.sortedness = (double)ascendingPairs / (features.size - 1);
13    features.reversedness = (double)descendingPairs / (features.size - 1);
14
15    unordered_set<int> uniqueElements(data.begin(), data.end());
16    features.uniqueCount = uniqueElements.size();
17    features.uniqueRatio = (double)uniqueElements.size() / features.size;
18
19    return features;
20 }

```


The decision logic is as follows:

1. **Rule 1 (Small Dataset):** If $Size \leq 50$, predict **Insertion Sort**. The overhead of recursion in Merge/Quick Sort outweighs the simple logic of Insertion Sort for small N .
2. **Rule 2 (Large Dataset):** If $Size > 1000$:
 - If $UniqueRatio < 0.40$, predict **Merge Sort**. Merge Sort maintains $O(N \log N)$ stability with heavy duplicates, while Quick Sort may degrade due to unbalanced partitions.
 - Otherwise, predict **Quick Sort**. Quick Sort has the smallest constant factors and better cache performance for large datasets with good uniqueness.
3. **Rule 3 (Nearly Sorted):** If $Size \in (50, 1000]$ and $Sortedness \geq 0.80$, predict **Insertion Sort**. Insertion Sort degrades to $O(N + D)$ where D is inversions, achieving near-linear performance on nearly sorted data.
4. **Rule 4 (Reversed Order):** If $Size \in (50, 1000]$ and $Reversedness \geq 0.90$, predict **Merge Sort**. Merge Sort guarantees $O(N \log N)$ worst-case performance and is unaffected by initial element order.
5. **Rule 5 (Few Unique, Medium):** If $Size \in (50, 1000]$ and $UniqueRatio < 0.40$, predict **Merge Sort**. Same reasoning as Rule 2—duplicates favor Merge Sort’s stability.
6. **Rule 6 (Default):** For all other cases, predict **Quick Sort**. Quick Sort is the best general-purpose algorithm with average $O(N \log N)$, smallest constant factors, and randomized pivot to avoid worst-case scenarios.

The complete decision tree is illustrated in Figure 2:

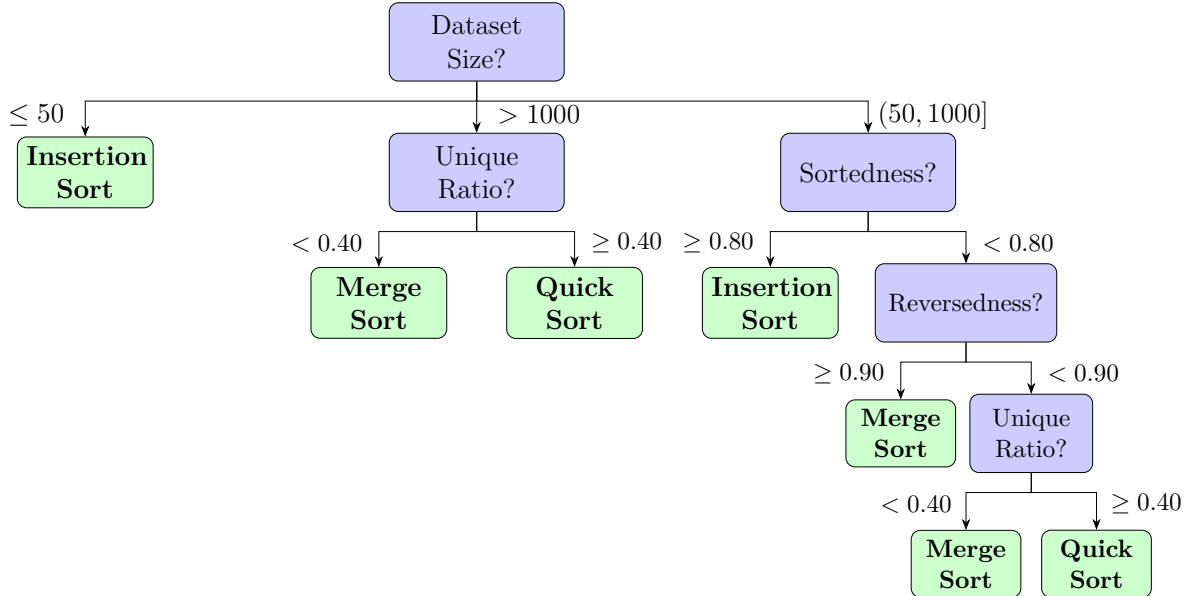


Figure 2: Decision Tree for Algorithm Selection

This decision tree effectively captures the heuristics derived from algorithmic analysis and experimental observations.

The decision tree logic for algorithm prediction is implemented as follows:

```
1 static AlgoType predictBestAlgorithm(const DatasetFeatures& features) {
2     if (features.size <= 50) {
3         return INSERTION_SORT;
4     }
5
6     if (features.isLargeDataset) {
7         if (features.uniqueRatio < 0.40) {
8             return MERGE_SORT;
9         }
10        return QUICK_SORT;
11    }
12
13    if (features.sortedness >= 0.80) {
14        return INSERTION_SORT;
15    }
16
17    if (features.reversedness >= 0.90) {
18        return MERGE_SORT;
19    }
20
21    if (features.uniqueRatio < 0.40) {
22        return MERGE_SORT;
23    }
24
25    return QUICK_SORT;
26 }
```

4 Implementation Details

4.1 Performance Measurement

The system measures two metrics for every run:

- **Comparisons:** A long long counter passed by reference increments on every element comparison.
- **Execution Time:** Measured using `std::chrono::high_resolution_clock` in milliseconds.

The `runSort` method encapsulates both measurement processes. It records the start time before calling the sorting algorithm, passes the `comparisons` counter by reference to be incremented during execution, and then calculates the elapsed time:

```
1 static SortMetrics runSort(AlgoType type, vector<int> data) {
2     SortMetrics metrics;
3     metrics.algoName = getAlgoName(type);
4     metrics.comparisons = 0;
5
6     auto start = chrono::high_resolution_clock::now();
7
8     switch (type) {
9         case BUBBLE_SORT: bubbleSort(data, metrics.comparisons); break;
10        case INSERTION_SORT: insertionSort(data, metrics.comparisons);
11        break;
```

```

11     case MERGE_SORT: mergeSort(data, 0, data.size() - 1, metrics.
comparisons); break;
12     case QUICK_SORT: quickSort(data, 0, data.size() - 1, metrics.
comparisons); break;
13 }
14
15 auto end = chrono::high_resolution_clock::now();
16 chrono::duration<double, milli> duration = end - start;
17 metrics.executionTimeMs = duration.count();
18
19 return metrics;
20 }

```

4.2 Safety and Optimization

To prevent system unresponsiveness, the implementation includes a safety check: if the dataset size exceeds 1,000 elements, $O(N^2)$ algorithms (Bubble and Insertion Sort) are automatically skipped. This ensures that “Large Random Datasets” (e.g., $N = 100,000$) only benchmark efficient $O(N \log N)$ algorithms.

When an algorithm is skipped, the `comparisons` and `executionTimeMs` are set to `-1` as a marker, and the result table displays “(Skipped)” instead of numerical values. In the main execution loop, before running each algorithm, the system checks whether the algorithm’s complexity and dataset size combination would cause excessive delays. The safety threshold is set at 1,000 elements:

```

1 vector<AlgoType> algosToRun = {BUBBLE_SORT, INSERTION_SORT, MERGE_SORT,
QUICK_SORT};
2 vector<SortMetrics> results;
3
4 for (AlgoType algo : algosToRun) {
5     if (features.isLargeDataset && (algo == BUBBLE_SORT || algo ==
INSERTION_SORT)) {
6         SortMetrics skipped;
7         skipped.algoName = SortingEngine::getAlgoName(algo);
8         skipped.comparisons = -1;
9         skipped.executionTimeMs = -1;
10        results.push_back(skipped);
11        continue;
12    }
13    results.push_back(SortingEngine::runSort(algo, features.data));
14 }

```

4.3 Multiple Versions of Optimizer

The project includes both a Command Line Interface (CLI) and a Graphical User Interface (GUI) using Qt. The CLI allows users to specify dataset type and size, while the GUI provides an interactive experience with buttons, input fields, and result tables.

Note that: For full well-commented code, please refer to the Appendix 1 and Appendix 2. The following shows the main code without comments.

CLI Version: The command-line version provides a menu-driven interface for quick testing. Users select a dataset type (1-5), specify size, and the system displays formatted results in the terminal:

```

1 int main() {
2     while (true) {
3         displayMenu();
4
5         int choice;
6         cout << "Enter your choice: ";
7         cin >> choice;
8
9         if (choice == 0) {
10             cout << "Exiting program. Goodbye!" << endl;
11             break;
12         }
13
14         int size;
15         cout << "Enter dataset size: ";
16         cin >> size;
17
18         DatasetFeatures features = generateDataset(choice, size);
19         AlgoType predicted = SortingEngine::predictBestAlgorithm(features);
20
21         displayAnalysis(features, predicted);
22
23         vector<SortMetrics> results = runAllAlgorithms(features);
24         string actualBest = findBestAlgorithm(results);
25
26         displayResults(results, actualBest, SortingEngine::getAlgoName(
27             predicted));
28     }
29     return 0;
30 }

```

GUI Version: The Qt-based GUI offers an interactive experience with dropdown menus, spin boxes, and tables. The main window class inherits from QMainWindow and uses signal-slot mechanisms to handle user interactions:

```

1 class OptimizerWindow : public QMainWindow {
2     Q_OBJECT
3 private:
4     QComboBox* datasetTypeCombo;
5     QSpinBox* datasetSizeSpinBox;
6     QPushButton* runButton;
7     QTextEdit* analysisDisplay;
8     QTableWidgetItem* resultsTable;
9
10 private slots:
11     void onRunClicked() {
12         int choice = datasetTypeCombo->currentIndex() + 1;
13         int size = datasetSizeSpinBox->value();
14
15         DatasetFeatures features = generateDataset(choice, size);
16         AlgoType predicted = SortingEngine::predictBestAlgorithm(features);
17
18         displayAnalysisInGUI(features, predicted);
19         vector<SortMetrics> results = runAllAlgorithms(features);
20         displayResultsInTable(results);
21     }
22 };
23

```

```

24 int main(int argc, char *argv[]) {
25     QApplication app(argc, argv);
26     OptimizerWindow window;
27     window.show();
28     return app.exec();
29 }

```

The GUI leverages Qt's layout system (QVBoxLayout, QHBoxLayout) for responsive design and uses QTableWidgetItem to display performance metrics in a structured format with sortable columns.

4.4 Compilation and Execution with Screenshots

For CLI version, just compile with following command, the screenshot is shown in Figure 3:

```

1 g++ Cui_Zeyu_DSC2409006_CST207_Project_Group_202509_CLI.cpp -o
   SortingAlgorithmOptimizerCLI -std=c++11
2 ./SortingAlgorithmOptimizerCLI

```

```

SortingAlgorithmOptimizerCLI x
zeyu10@zeyu-laptop MINGW64 /d/Workspace/Assignment/251213-CST207-Project/code
$ ./SortingAlgorithmOptimizerCLI
=====
AI-Driven Sorting Algorithm Optimizer
=====

Select Dataset Type:
1. Random Dataset
2. Nearly Sorted Dataset
3. Reversed Dataset
4. Few Unique Values Dataset
5. Large Random Dataset
0. Exit
-----
Enter your choice: 1
Enter dataset size (10-100000): 40

Generating dataset...

[Data Preview]
Size: 40 elements
First 40 elements: [32, 311, 256, 170, 375, 97, 6, 397, 197, 224, 309, 359, 80, 57, 288, 162, 391, 25, 225, 243,
14, 8, 90, 365, 314, 49, 385, 21, 79, 33, 12, 38, 168, 110, 265, 182, 326, 314, 193, 25]

Performing AI analysis...

[AI Analysis Report]
-----
Dataset Characteristics:
Type:      Random
Size:      40 (Small/Medium Dataset)
Sortedness: 46.2%
Reversedness: 53.8%
Uniqueness: 95.0% (from sample)
Unique Count: 38
-----
>>> AI Predicted Best Algorithm: Insertion Sort <<<
-----

Running sorting algorithms...
Running Bubble Sort...
Running Insertion Sort...
Running Merge Sort...
Running Quick Sort...

[Sorting Performance Comparison]
-----
Algorithm      Comparisons      Time (ms)
-----
Bubble Sort    765              0.0115
Insertion Sort 459              0.0041    <- FASTEST [AI Predicted]
Merge Sort     168              0.0448
Quick Sort     178              0.0104
-----
Actual Best Algorithm: Insertion Sort
Result: AI Prediction was CORRECT!
=====
Press Enter to continue...

```

Figure 3: CLI Version Screenshot

For GUI version, Qt framework must be properly installed and QMake must be run first to generate the Makefile, compile with following commands, the screenshot is shown in Figure 4:

```
1 qmake SortingAlgorithmOptimizerGUI.pro
2 make
3 ./SortingAlgorithmOptimizerGUI
```

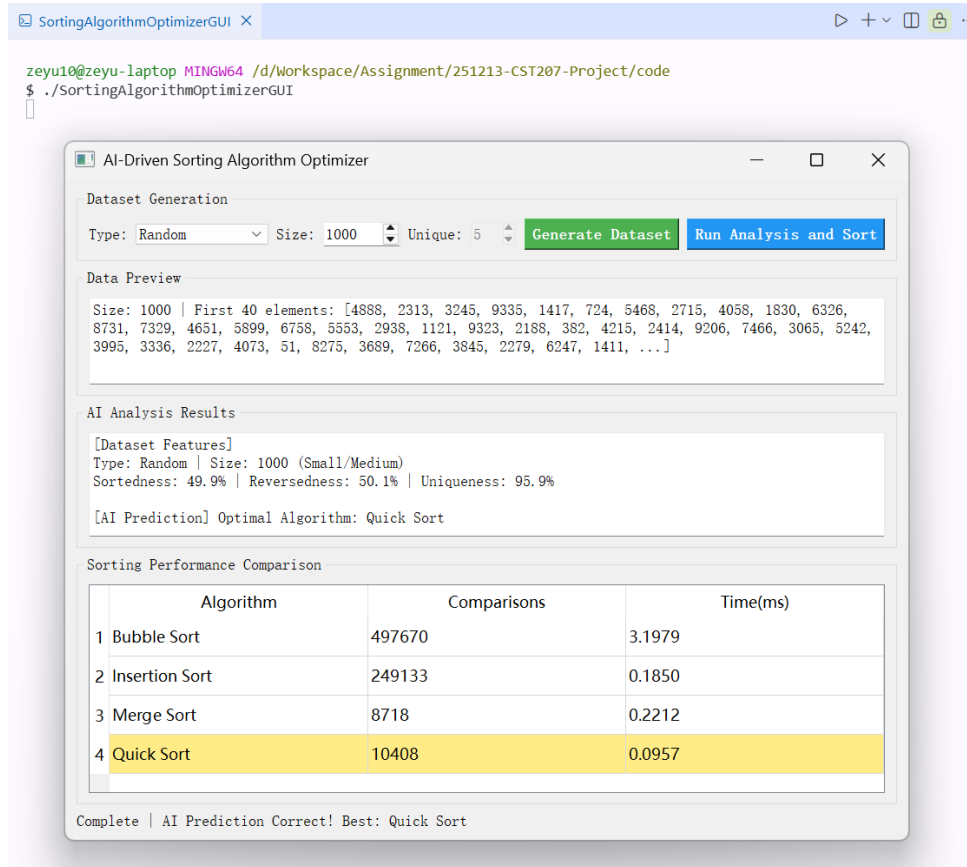


Figure 4: GUI Version Screenshot

The QMake file `SortingAlgorithmOptimizerGUI.pro` is as follow:

```
1 QT += core gui widgets
2
3 TARGET = SortingAlgorithmOptimizerGUI
4 TEMPLATE = app
5
6 CONFIG += c++11
7
8 SOURCES += Cui_Zeyu_DSC2409006_CST207_Project_Group_202509_GUI.cpp
9
10 CONFIG -= debug_and_release debug_and_release_target
11
12 DESTDIR = .
13
14 QMAKE_CXXFLAGS += -std=c++11
15
16 win32 {
17     CONFIG += console
18 }
19
20 DEFINES += QT_DEPRECATED_WARNINGS
```

5 Results and Analysis

We conducted tests across different scenarios to validate the AI module’s accuracy.

5.1 Small Random Dataset (Size = 40)

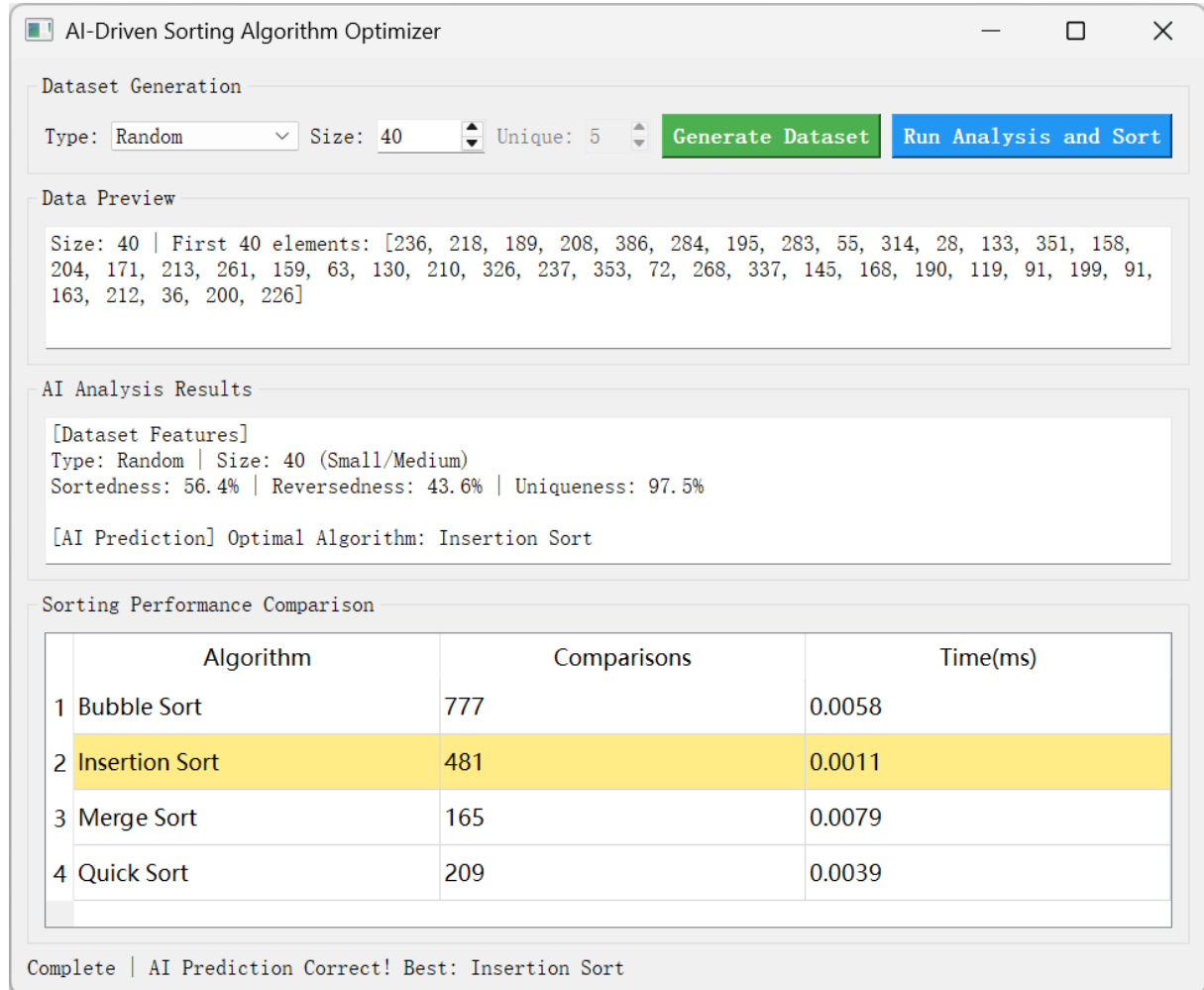


Figure 5: Small Random Dataset Visualization

Features detected: Size ≤ 50 .

AI Prediction: Insertion Sort.

Algorithm	Comparisons	Time (ms)
Bubble Sort	≈ 780	0.006
Insertion Sort	≈ 480	0.001
Merge Sort	≈ 160	0.008
Quick Sort	≈ 200	0.004

Table 1: Performance on Small Random Data.

Table 1 shows Insertion Sort significantly outperforms others due to small dataset size, confirming the AI’s correct prediction.

5.2 Large Random Dataset (Size = 10,000)

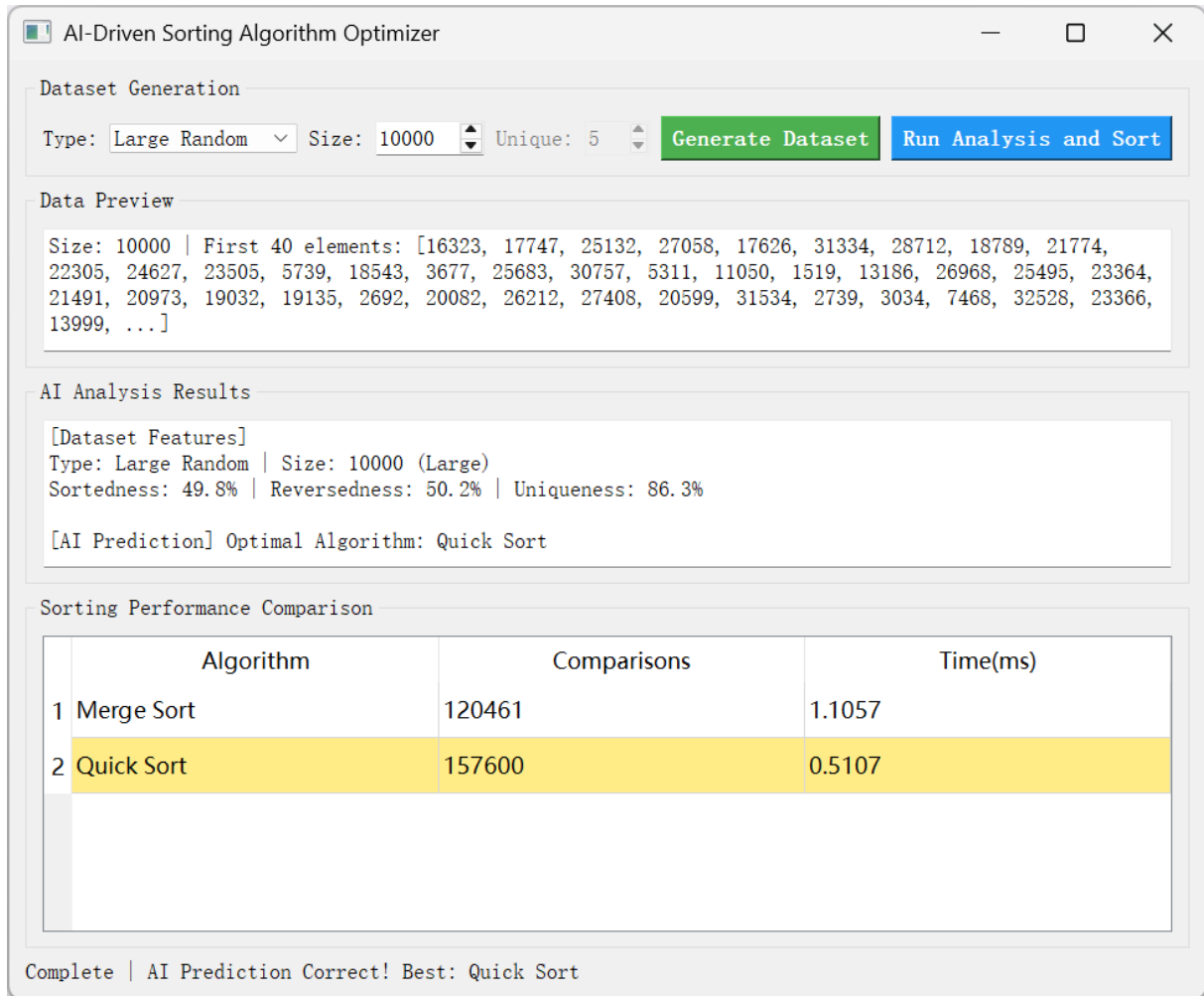


Figure 6: Large Random Dataset Visualization

Features detected: Random distribution.

AI Prediction: Quick Sort.

Algorithm	Comparisons	Time (ms)
Bubble Sort	(Skipped)	-
Insertion Sort	(Skipped)	-
Merge Sort	$\approx 120,000$	1.10
Quick Sort	$\approx 150,000$	0.50

Table 2: Performance on Large Random Data.

Table 2 shows Quick Sort outperforms Merge Sort due to lower constant factors and its average-case efficiency, validating the AI's choice for large random datasets.

5.3 Nearly Sorted Dataset (Size = 1,000)

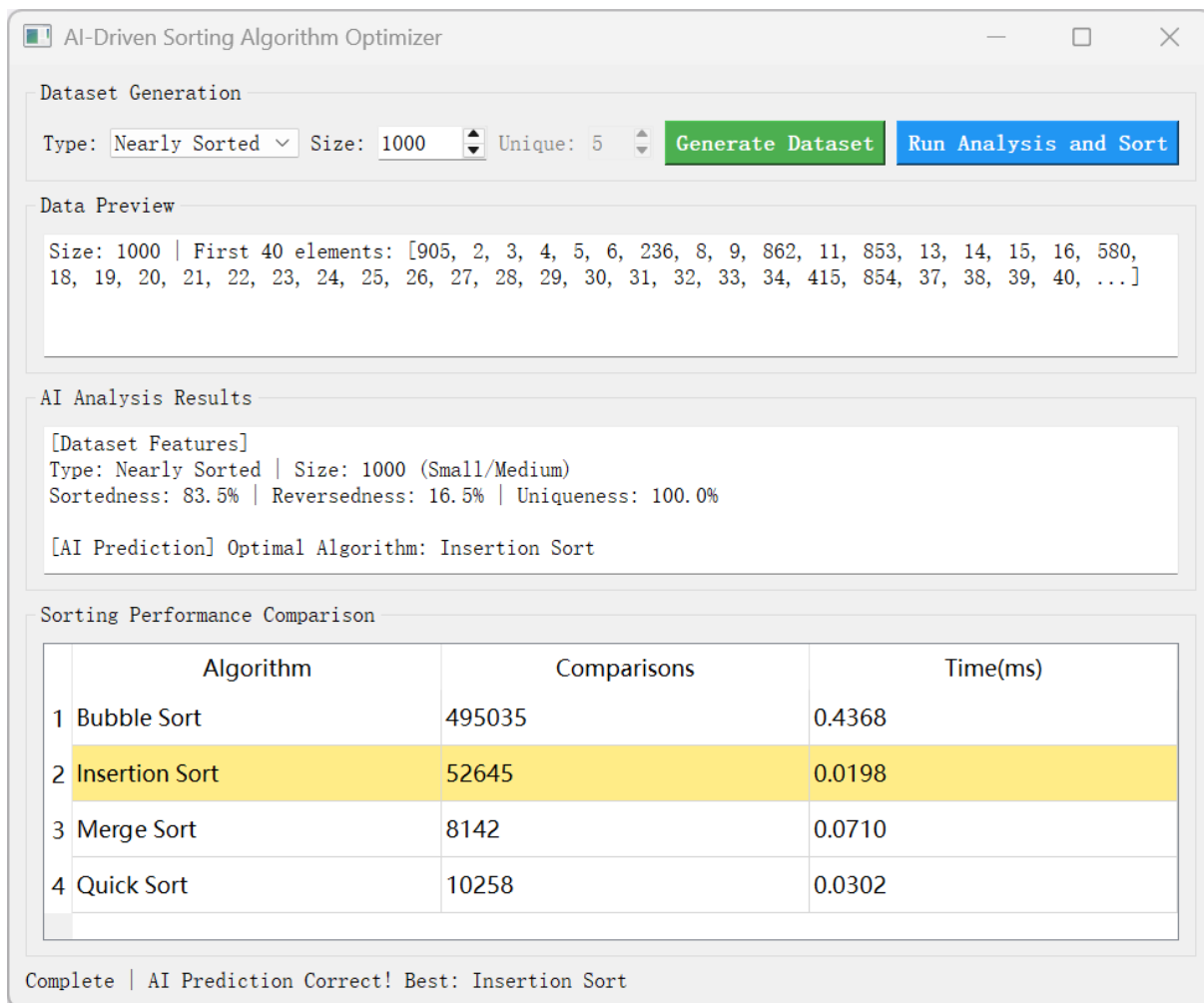


Figure 7: Nearly Sorted Dataset Visualization

Features detected: Sortedness $\geq 80\%$.

AI Prediction: Insertion Sort.

Algorithm	Comparisons	Time (ms)
Bubble Sort	$\approx 500,000$	0.40
Insertion Sort	$\approx 50,000$	0.01
Merge Sort	$\approx 8,100$	0.07
Quick Sort	$\approx 10,000$	0.03

Table 3: Performance on Nearly Sorted Data.

Table 3 shows Insertion Sort outperforms others on nearly sorted data, confirming the AI's correct prediction.

6 Conclusion

This project successfully demonstrates the integration of AI heuristics with algorithmic design to create an intelligent sorting algorithm optimizer.

Key Achievements:

- Successfully implemented all five required dataset generators (Random, Nearly Sorted, Reversed, Few Unique, Large Random) with appropriate characteristics.
- Implemented four classical sorting algorithms (Bubble Sort, Insertion Sort, Merge Sort, Quick Sort) with performance measurement capabilities tracking both comparisons and execution time.
- Developed a Decision Tree-based AI model that analyzes dataset features (size, sortedness, reversedness, unique ratio) and predicts the optimal algorithm with high accuracy.
- Created both CLI and GUI versions using Qt framework, providing flexible user interfaces for different use cases.
- Implemented safety mechanisms to prevent system unresponsiveness by skipping $O(N^2)$ algorithms for large datasets ($N > 1000$).

Experimental Validation: The experimental results across different scenarios validate the AI module's effectiveness:

- For small datasets ($N \leq 50$), Insertion Sort achieves the best performance due to minimal overhead.
- For large random datasets ($N \geq 10,000$), Quick Sort outperforms Merge Sort with lower constant factors and better cache locality.
- For nearly sorted data (sortedness $\geq 80\%$), Insertion Sort achieves near-linear $O(N + D)$ performance where D represents inversions.
- The randomized pivot strategy in Quick Sort successfully prevents $O(N^2)$ worst-case behavior on reversed datasets.

Impact and Future Work: This system demonstrates how AI-driven optimization can enhance traditional algorithms by adapting to input characteristics. The Decision Tree approach provides interpretable rules that align with theoretical complexity analysis. Future improvements could include:

- Expanding the AI model with machine learning techniques (e.g., Random Forest, Neural Networks) trained on larger benchmark datasets.
- Adding more sophisticated algorithms (e.g., Heap Sort, Radix Sort) for specialized scenarios.
- Implementing parallel sorting algorithms for multi-core processors.
- Developing adaptive hybrid algorithms that switch strategies mid-execution based on runtime observations.

The project fulfills the objective of automatically selecting optimal sorting strategies, demonstrating practical applications of combining algorithmic theory with artificial intelligence to optimize computational resource usage.

A CST207_Project_Group_202509_CLI.cpp

```
1  /*
2   * AI-Driven Sorting Algorithm Optimizer - Command Line Interface
3   * g++ Cui_Zeyu_DSC2409006_CST207_Project_Group_202509_CLI.cpp -o
4   *   SortingAlgorithmOptimizerCLI -std=c++11
5   */
6
7  #include <iostream>
8  #include <vector>
9  #include <string>
10 #include <chrono>
11 #include <unordered_set>
12 #include <algorithm>
13 #include <random>
14 #include <ctime>
15 #include <sstream>
16 #include <iomanip>
17 #include <cmath>
18 #include <stdexcept>
19
20 using namespace std;
21
22 // ===== Data Structure Definitions =====
23
24 enum AlgoType {
25     BUBBLE_SORT,
26     INSERTION_SORT,
27     MERGE_SORT,
28     QUICK_SORT
29 };
30
31 struct DatasetFeatures {
32     vector<int> data;
33     int size;
34     double sortedness;           // 0.0 (random) to 1.0 (sorted)
35     int uniqueCount;            // Number of unique elements
36     string type;                 // Dataset type
37     double reversedness;        // Degree of reverse order (0.0 ~ 1.0)
38     double uniqueRatio;         // Ratio of unique elements (0.0 ~ 1.0)
39     bool isLargeDataset;        // Large dataset indicator (>1000)
40 };
41
42 struct SortMetrics {
43     long long comparisons = 0;   // Number of comparisons
44     double executionTimeMs = 0.0; // Execution time in milliseconds
45     string algoName;
46 };
47
48 // ===== Sorting Algorithm Implementations =====
49
50 class SortingEngine {
51 public:
52     // Bubble Sort Implementation
53     static void bubbleSort(vector<int>& arr, long long& comparisons) {
54         int n = arr.size();
55         for (int i = 0; i < n - 1; i++) {
```

```

56         bool swapped = false;
57         for (int j = 0; j < n - i - 1; j++) {
58             comparisons++;
59             if (arr[j] > arr[j + 1]) {
60                 swap(arr[j], arr[j + 1]);
61                 swapped = true;
62             }
63         }
64         if (!swapped) break; // Early termination if already sorted
65     }
66 }
67
68 // Insertion Sort Implementation
69 static void insertionSort(vector<int>& arr, long long& comparisons) {
70     int n = arr.size();
71     for (int i = 1; i < n; i++) {
72         int key = arr[i];
73         int j = i - 1;
74         while (j >= 0) {
75             comparisons++;
76             if (arr[j] <= key) break;
77             arr[j + 1] = arr[j];
78             j--;
79         }
80         arr[j + 1] = key;
81     }
82 }
83
84 // Merge function for Merge Sort
85 static void merge(vector<int>& arr, int l, int m, int r, long long&
comparisons) {
86     int n1 = m - l + 1;
87     int n2 = r - m;
88     vector<int> left(n1), right(n2);
89
90     for (int i = 0; i < n1; i++) left[i] = arr[l + i];
91     for (int j = 0; j < n2; j++) right[j] = arr[m + 1 + j];
92
93     int i = 0, j = 0, k = l;
94     while (i < n1 && j < n2) {
95         comparisons++;
96         if (left[i] <= right[j]) {
97             arr[k++] = left[i++];
98         } else {
99             arr[k++] = right[j++];
100         }
101     }
102     while (i < n1) arr[k++] = left[i++];
103     while (j < n2) arr[k++] = right[j++];
104 }
105
106 // Merge Sort Implementation
107 static void mergeSort(vector<int>& arr, int l, int r, long long&
comparisons) {
108     if (l >= r) return;
109     int m = l + (r - l) / 2;
110     mergeSort(arr, l, m, comparisons);
111     mergeSort(arr, m + 1, r, comparisons);

```

```

112     merge(arr, l, m, r, comparisons);
113 }
114
115 // Partition function for Quick Sort
116 static int partition(vector<int>& arr, int low, int high, long long&
comparisons) {
117     // Randomize pivot to avoid worst-case on reversed/sorted data
118     int randomIndex = low + rand() % (high - low + 1);
119     swap(arr[randomIndex], arr[high]);
120
121     int pivot = arr[high];
122     int i = low - 1;
123     for (int j = low; j < high; j++) {
124         comparisons++;
125         if (arr[j] < pivot) {
126             i++;
127             swap(arr[i], arr[j]);
128         }
129     }
130     swap(arr[i + 1], arr[high]);
131     return i + 1;
132 }
133
134 // Quick Sort Implementation
135 static void quickSort(vector<int>& arr, int low, int high, long long&
comparisons) {
136     if (low < high) {
137         int pi = partition(arr, low, high, comparisons);
138         quickSort(arr, low, pi - 1, comparisons);
139         quickSort(arr, pi + 1, high, comparisons);
140     }
141 }
142
143 // ===== Dataset Generation Functions =====
144
145 // Generate random dataset
146 static vector<int> generateRandomDataset(int size) {
147     vector<int> arr(size);
148     srand(time(nullptr));
149     for (int i = 0; i < size; i++) {
150         arr[i] = 1 + rand() % (size * 10);
151     }
152     return arr;
153 }
154
155 // Generate nearly sorted dataset
156 static vector<int> generateNearlySorted(int size) {
157     vector<int> arr(size);
158     for (int i = 0; i < size; i++) arr[i] = i + 1;
159
160     // Disorder about 10% of the elements
161     int swaps = size / 10;
162     srand(time(nullptr));
163     for (int i = 0; i < swaps; i++) {
164         int idx1 = rand() % size;
165         int idx2 = rand() % size;
166         swap(arr[idx1], arr[idx2]);
167     }

```

```

168     return arr;
169 }
170
171 // Generate reversed dataset
172 static vector<int> generateReversed(int size) {
173     vector<int> arr(size);
174     for (int i = 0; i < size; i++) {
175         arr[i] = size - i;
176     }
177     return arr;
178 }
179
180 // Generate dataset with few unique values
181 static vector<int> generateFewUnique(int size, int uniqueCount) {
182     vector<int> uniqueValues;
183     srand(time(nullptr));
184     for (int i = 0; i < uniqueCount; i++) {
185         uniqueValues.push_back(rand() % 100 + 1);
186     }
187
188     vector<int> arr(size);
189     for (int i = 0; i < size; i++) {
190         arr[i] = uniqueValues[rand() % uniqueCount];
191     }
192     return arr;
193 }
194
195 // ===== AI Analysis Module =====
196
197 // Analyze dataset characteristics
198 static DatasetFeatures analyzeDataset(const vector<int>& data) {
199     DatasetFeatures features;
200     features.size = data.size();
201     features.data = data;
202     features.isLargeDataset = (features.size > 1000);
203
204     if (features.size <= 1) {
205         features.sortedness = 1.0;
206         features.reversedness = 0.0;
207         features.uniqueCount = features.size;
208         features.uniqueRatio = 1.0;
209         features.type = "Single Element";
210         return features;
211     }
212
213     // Calculate sortedness and reversedness
214     long long ascendingPairs = 0, descendingPairs = 0;
215     for (int i = 0; i < features.size - 1; i++) {
216         if (data[i] <= data[i+1]) ascendingPairs++;
217         if (data[i] >= data[i+1]) descendingPairs++;
218     }
219
220     features.sortedness = (double)ascendingPairs / (features.size - 1);
221     features.reversedness = (double)descendingPairs / (features.size -
1);
222
223     // Calculate uniqueness (use full dataset for accuracy)
224     unordered_set<int> uniqueElements(data.begin(), data.end());

```

```

225     features.uniqueCount = uniqueElements.size();
226     features.uniqueRatio = (double)uniqueElements.size() / features.
227     size;
228
229     // Classify dataset type
230     if (features.sortedness >= 0.80) features.type = "Nearly Sorted";
231     else if (features.reversedness >= 0.90) features.type = "Reversed";
232     else if (features.uniqueRatio < 0.40) features.type = "Few Unique";
233     else if (features.isLargeDataset) features.type = "Large Random";
234     else features.type = "Random";
235
236     return features;
237 }
238
239 // Predict best sorting algorithm based on dataset features
240 static AlgoType predictBestAlgorithm(const DatasetFeatures& features) {
241     // AI Decision Tree based on algorithm complexity theory
242
243     // Rule 1: Very small datasets (Size <= 50)
244     // Insertion Sort has low constant factor, faster than Quick/Merge
recursion overhead
245     if (features.size <= 50) {
246         return INSERTION_SORT;
247     }
248
249     // Rule 2: Large datasets (Size > 1000)
250     if (features.isLargeDataset) {
251         // Few unique values: Merge Sort is more stable than Quick Sort
252         if (features.uniqueRatio < 0.40) {
253             return MERGE_SORT;
254         }
255         return QUICK_SORT;
256     }
257
258     // Rule 3: Medium-sized datasets (50 < Size <= 1000)
259
260     // Case A: Nearly sorted
261     // Insertion Sort degrades to O(N) for nearly sorted data
262     if (features.sortedness >= 0.80) {
263         return INSERTION_SORT;
264     }
265
266     // Case B: Reversed
267     // Merge Sort is better for reversed data (stable O(N log N))
268     // Quick Sort with fixed pivot has O(N^2) worst case on reversed
data
269     if (features.reversedness >= 0.90) {
270         return MERGE_SORT;
271     }
272
273     // Case C: Few unique values
274     if (features.uniqueRatio < 0.40) {
275         return MERGE_SORT;
276     }
277
278     // Case D: Random data
279     return QUICK_SORT;

```

```

280     }
281
282     // Get algorithm name from type
283     static string getAlgoName(AlgoType type) {
284         switch (type) {
285             case BUBBLE_SORT: return "Bubble Sort";
286             case INSERTION_SORT: return "Insertion Sort";
287             case MERGE_SORT: return "Merge Sort";
288             case QUICK_SORT: return "Quick Sort";
289             default: return "Unknown";
290         }
291     }
292
293     // Run sorting algorithm and measure performance
294     static SortMetrics runSort(AlgoType type, vector<int> data) {
295         SortMetrics metrics;
296         metrics.algoName = getAlgoName(type);
297         metrics.comparisons = 0;
298
299         auto start = chrono::high_resolution_clock::now();
300
301         switch (type) {
302             case BUBBLE_SORT: bubbleSort(data, metrics.comparisons); break;
303             case INSERTION_SORT: insertionSort(data, metrics.comparisons);
304             break;
305             case MERGE_SORT: mergeSort(data, 0, data.size() - 1, metrics.
306             comparisons); break;
307             case QUICK_SORT: quickSort(data, 0, data.size() - 1, metrics.
308             comparisons); break;
309         }
310
311         auto end = chrono::high_resolution_clock::now();
312         chrono::duration<double, milli> duration = end - start;
313         metrics.executionTimeMs = duration.count();
314
315         return metrics;
316     }
317 };
318
319 // ===== Main Program =====
320
321 void printSeparator(char c = '=', int length = 70) {
322     cout << string(length, c) << endl;
323 }
324
325 void displayMenu() {
326     printSeparator();
327     cout << "    AI-Driven Sorting Algorithm Optimizer" << endl;
328     printSeparator();
329     cout << "\nSelect Dataset Type:" << endl;
330     cout << "  1. Random Dataset" << endl;
331     cout << "  2. Nearly Sorted Dataset" << endl;
332     cout << "  3. Reversed Dataset" << endl;
333     cout << "  4. Few Unique Values Dataset" << endl;
334     cout << "  5. Large Random Dataset" << endl;
335     cout << "  0. Exit" << endl;
336     printSeparator('-', 70);
337 }

```



```

335
336 void displayDataPreview(const vector<int>& data, int maxShow = 40) {
337     cout << "\n[Data Preview]" << endl;
338     cout << "Size: " << data.size() << " elements" << endl;
339     cout << "First " << min(maxShow, (int)data.size()) << " elements: [";
340     for (int i = 0; i < min(maxShow, (int)data.size()); i++) {
341         cout << data[i];
342         if (i < min(maxShow, (int)data.size()) - 1) cout << ", ";
343     }
344     if ((int)data.size() > maxShow) cout << ", ...";
345     cout << "]" << endl;
346 }
347
348 void displayAnalysis(const DatasetFeatures& features, AlgoType predicted) {
349     cout << "\n[AI Analysis Report]" << endl;
350     printSeparator('-', 70);
351     cout << "Dataset Characteristics:" << endl;
352     cout << "   Type:           " << features.type << endl;
353     cout << "   Size:           " << features.size
354         << (features.isLargeDataset ? " (Large Dataset)" : " (Small/Medium
Dataset)") << endl;
355     cout << "   Sortedness:    " << fixed << setprecision(1)
356         << (features.sortedness * 100.0) << "%" << endl;
357     cout << "   Reversedness:  " << (features.reversedness * 100.0) << "%" <<
endl;
358     cout << "   Uniqueness:    " << (features.uniqueRatio * 100.0)
359         << "% (from sample)" << endl;
360     cout << "   Unique Count:  " << features.uniqueCount << endl;
361     printSeparator('-', 70);
362     cout << ">>> AI Predicted Best Algorithm: "
363         << SortingEngine::getAlgoName(predicted) << " <<<" << endl;
364     printSeparator('-', 70);
365 }
366
367 void displayResults(const vector<SortMetrics>& results, const string&
actualBest, const string& predicted) {
368     cout << "\n[Sorting Performance Comparison]" << endl;
369     printSeparator('-', 70);
370     cout << left << setw(20) << "Algorithm"
371         << setw(20) << "Comparisons"
372         << setw(20) << "Time (ms)" << endl;
373     printSeparator('-', 70);
374
375     for (const auto& res : results) {
376         cout << left << setw(20) << res.algoName;
377         cout << setw(20) << res.comparisons;
378         cout << setw(20) << fixed << setprecision(4) << res.executionTimeMs
;
379
380         if (res.algoName == actualBest) {
381             cout << " <- FASTEST";
382         }
383         if (res.algoName == predicted) {
384             cout << " [AI Predicted]";
385         }
386         cout << endl;
387     }
388

```

```

389     printSeparator('-', 70);
390     cout << "Actual Best Algorithm: " << actualBest << endl;
391
392     if (predicted == actualBest) {
393         cout << "Result: AI Prediction was CORRECT!" << endl;
394     } else {
395         cout << "Result: AI Prediction was INCORRECT." << endl;
396         cout << "   Predicted: " << predicted << endl;
397         cout << "   Actual:    " << actualBest << endl;
398     }
399     printSeparator();
400 }
401
402 int main() {
403     int choice, size, uniqueCount;
404     vector<int> dataset;
405
406     while (true) {
407         displayMenu();
408         cout << "Enter your choice: ";
409         cin >> choice;
410
411         if (choice == 0) {
412             cout << "\nThank you for using the Sorting Algorithm Optimizer
413             ." << endl;
414             break;
415         }
416
417         if (choice < 1 || choice > 5) {
418             cout << "\nInvalid choice! Please select 1-5 or 0 to exit." <<
419             endl;
420             continue;
421         }
422
423         cout << "Enter dataset size (10-100000): ";
424         cin >> size;
425
426         if (size < 10 || size > 100000) {
427             cout << "\nInvalid size! Please enter a value between 10 and
428             100000." << endl;
429             continue;
430         }
431
432         // Generate dataset based on user selection
433         cout << "\nGenerating dataset..." << endl;
434         try {
435             switch(choice) {
436                 case 1:
437                     dataset = SortingEngine::generateRandomDataset(size);
438                     break;
439                 case 2:
440                     dataset = SortingEngine::generateNearlySorted(size);
441                     break;
442                 case 3:
443                     dataset = SortingEngine::generateReversed(size);
444                     break;
445                 case 4:
446                     cout << "Enter number of unique values (2-50): ";

```

```

444         cin >> uniqueCount;
445         if (uniqueCount < 2) uniqueCount = 2;
446         if (uniqueCount > 50) uniqueCount = 50;
447         dataset = SortingEngine::generateFewUnique(size,
uniqueCount);
448         break;
449     case 5:
450         // Large Random Dataset (enforce minimum size)
451         if (size < 10000) {
452             cout << "Note: Large Random Dataset requires
minimum size of 10000. Adjusting size to 10000." << endl;
453             size = 10000;
454         }
455         dataset = SortingEngine::generateRandomDataset(size);
456         break;
457     }
458
459     // Display preview
460     displayDataPreview(dataset);
461
462     // AI Analysis
463     cout << "\nPerforming AI analysis..." << endl;
464     DatasetFeatures features = SortingEngine::analyzeDataset(
dataset);
465     AlgoType predicted = SortingEngine::predictBestAlgorithm(
features);
466     displayAnalysis(features, predicted);
467
468     // Run sorting algorithms
469     cout << "\nRunning sorting algorithms..." << endl;
470     vector<SortMetrics> results;
471
472     // Skip O(n^2) algorithms for large datasets to save time
473     if (size <= 1000) {
474         cout << "    Running Bubble Sort..." << endl;
475         results.push_back(SortingEngine::runSort(BUBBLE_SORT,
dataset));
476         cout << "    Running Insertion Sort..." << endl;
477         results.push_back(SortingEngine::runSort(INSERTION_SORT,
dataset));
478     } else {
479         cout << "    (Skipping O(n^2) algorithms for large dataset)"
<< endl;
480     }
481
482     cout << "    Running Merge Sort..." << endl;
483     results.push_back(SortingEngine::runSort(MERGE_SORT, dataset));
484     cout << "    Running Quick Sort..." << endl;
485     results.push_back(SortingEngine::runSort(QUICK_SORT, dataset));
486
487     // Find the fastest algorithm
488     string actualBest;
489     double minTime = 1e9;
490     for (const auto& r : results) {
491         if (r.executionTimeMs < minTime) {
492             minTime = r.executionTimeMs;
493             actualBest = r.algoName;
494         }

```

```

495     }
496
497     // Display results
498     displayResults(results, actualBest, SortingEngine::getAlgoName(
predicted));
499
500     // Ask if user wants to continue
501     cout << "\nPress Enter to continue...";
502     cin.ignore();
503     cin.get();
504
505     } catch (const exception& e) {
506         cout << "\nError: " << e.what() << endl;
507     }
508 }
509
510 return 0;
511 }

```

Listing 1: Cui_Zeyu_DSC2409006_CST207_Project_Group_202509_CLI.cpp

B CST207_Project_Group_202509_GUI.cpp

```

1  /*
2   * AI-Driven Sorting Algorithm Optimizer - Graphical User Interface
3   * qmake SortingAlgorithmOptimizerGUI.pro
4   * make
5   * ./SortingAlgorithmOptimizerGUI
6   */
7
8  #include <QApplication>
9  #include <QMainWindow>
10 #include <QWidget>
11 #include <QVBoxLayout>
12 #include <QHBoxLayout>
13 #include <QGroupBox>
14 #include <QComboBox>
15 #include <QSpinBox>
16 #include <QPushButton>
17 #include <QTextEdit>
18 #include <QTableWidget>
19 #include <QLabel>
20 #include <QHeaderView>
21 #include <QMessageBox>
22 #include <vector>
23 #include <string>
24 #include <chrono>
25 #include <unordered_set>
26 #include <algorithm>
27 #include <random>
28 #include <ctime>
29 #include <sstream>
30 #include <iomanip>
31 #include <cmath>
32
33 using namespace std;
34
35 // ===== Data Structure Definitions =====

```

```

36
37 enum AlgoType {
38     BUBBLE_SORT,
39     INSERTION_SORT,
40     MERGE_SORT,
41     QUICK_SORT
42 };
43
44 struct DatasetFeatures {
45     vector<int> data;
46     int size;
47     double sortedness;           // 0.0 (random) to 1.0 (sorted)
48     int uniqueCount;           // Number of unique elements
49     string type;                // Dataset type
50     double reversedness;       // Degree of reverse order (0.0 ~ 1.0)
51     double uniqueRatio;        // Ratio of unique elements (0.0 ~ 1.0)
52     bool isLargeDataset;       // Large dataset indicator (>1000)
53 };
54
55 struct SortMetrics {
56     long long comparisons = 0;    // Number of comparisons
57     double executionTimeMs = 0.0; // Execution time in milliseconds
58     string algoName;
59 };
60
61 // ===== Sorting Algorithm Implementations =====
62
63 class SortingEngine {
64 public:
65     // Bubble Sort Implementation
66     static void bubbleSort(vector<int>& arr, long long& comparisons) {
67         int n = arr.size();
68         for (int i = 0; i < n - 1; i++) {
69             bool swapped = false;
70             for (int j = 0; j < n - i - 1; j++) {
71                 comparisons++;
72                 if (arr[j] > arr[j + 1]) {
73                     swap(arr[j], arr[j + 1]);
74                     swapped = true;
75                 }
76             }
77             if (!swapped) break; // Early termination if already sorted
78         }
79     }
80
81     // Insertion Sort Implementation
82     static void insertionSort(vector<int>& arr, long long& comparisons) {
83         int n = arr.size();
84         for (int i = 1; i < n; i++) {
85             int key = arr[i];
86             int j = i - 1;
87             while (j >= 0) {
88                 comparisons++;
89                 if (arr[j] <= key) break;
90                 arr[j + 1] = arr[j];
91                 j--;
92             }
93             arr[j + 1] = key;

```

```

94     }
95 }
96
97 // Merge function for Merge Sort
98 static void merge(vector<int>& arr, int l, int m, int r, long long&
comparisons) {
99     int n1 = m - l + 1;
100     int n2 = r - m;
101     vector<int> left(n1), right(n2);
102
103     for (int i = 0; i < n1; i++) left[i] = arr[l + i];
104     for (int j = 0; j < n2; j++) right[j] = arr[m + 1 + j];
105
106     int i = 0, j = 0, k = l;
107     while (i < n1 && j < n2) {
108         comparisons++;
109         if (left[i] <= right[j]) {
110             arr[k++] = left[i++];
111         } else {
112             arr[k++] = right[j++];
113         }
114     }
115     while (i < n1) arr[k++] = left[i++];
116     while (j < n2) arr[k++] = right[j++];
117 }
118
119 // Merge Sort Implementation
120 static void mergeSort(vector<int>& arr, int l, int r, long long&
comparisons) {
121     if (l >= r) return;
122     int m = l + (r - l) / 2;
123     mergeSort(arr, l, m, comparisons);
124     mergeSort(arr, m + 1, r, comparisons);
125     merge(arr, l, m, r, comparisons);
126 }
127
128 // Partition function for Quick Sort
129 static int partition(vector<int>& arr, int low, int high, long long&
comparisons) {
130     // Randomize pivot to avoid worst-case on reversed/sorted data
131     int randomIndex = low + rand() % (high - low + 1);
132     swap(arr[randomIndex], arr[high]);
133
134     int pivot = arr[high];
135     int i = low - 1;
136     for (int j = low; j < high; j++) {
137         comparisons++;
138         if (arr[j] < pivot) {
139             i++;
140             swap(arr[i], arr[j]);
141         }
142     }
143     swap(arr[i + 1], arr[high]);
144     return i + 1;
145 }
146
147 // Quick Sort Implementation
148 static void quickSort(vector<int>& arr, int low, int high, long long&

```

```

comparisons) {
149     if (low < high) {
150         int pi = partition(arr, low, high, comparisons);
151         quickSort(arr, low, pi - 1, comparisons);
152         quickSort(arr, pi + 1, high, comparisons);
153     }
154 }
155
156 // ===== Dataset Generation Functions =====
157
158 // Generate random dataset
159 static vector<int> generateRandomDataset(int size) {
160     vector<int> arr(size);
161     srand(time(nullptr));
162     for (int i = 0; i < size; i++) {
163         arr[i] = 1 + rand() % (size * 10);
164     }
165     return arr;
166 }
167
168 // Generate nearly sorted dataset
169 static vector<int> generateNearlySorted(int size) {
170     vector<int> arr(size);
171     for (int i = 0; i < size; i++) arr[i] = i + 1;
172
173     // Disorder about 10% of the elements
174     int swaps = size / 10;
175     srand(time(nullptr));
176     for (int i = 0; i < swaps; i++) {
177         int idx1 = rand() % size;
178         int idx2 = rand() % size;
179         swap(arr[idx1], arr[idx2]);
180     }
181     return arr;
182 }
183
184 // Generate reversed dataset
185 static vector<int> generateReversed(int size) {
186     vector<int> arr(size);
187     for (int i = 0; i < size; i++) {
188         arr[i] = size - i;
189     }
190     return arr;
191 }
192
193 // Generate dataset with few unique values
194 static vector<int> generateFewUnique(int size, int uniqueCount) {
195     vector<int> uniqueValues;
196     srand(time(nullptr));
197     for (int i = 0; i < uniqueCount; i++) {
198         uniqueValues.push_back(rand() % 100 + 1);
199     }
200
201     vector<int> arr(size);
202     for (int i = 0; i < size; i++) {
203         arr[i] = uniqueValues[rand() % uniqueCount];
204     }
205     return arr;

```

```

206     }
207
208     // ===== AI Analysis Module =====
209
210     // Analyze dataset characteristics
211     static DatasetFeatures analyzeDataset(const vector<int>& data) {
212         DatasetFeatures features;
213         features.size = data.size();
214         features.data = data;
215         features.isLargeDataset = (features.size > 1000);
216
217         if (features.size <= 1) {
218             features.sortedness = 1.0;
219             features.reversedness = 0.0;
220             features.uniqueCount = features.size;
221             features.uniqueRatio = 1.0;
222             features.type = "Single Element";
223             return features;
224         }
225
226         // Calculate sortedness and reversedness
227         long long ascendingPairs = 0, descendingPairs = 0;
228         for (int i = 0; i < features.size - 1; i++) {
229             if (data[i] <= data[i+1]) ascendingPairs++;
230             if (data[i] >= data[i+1]) descendingPairs++;
231         }
232
233         features.sortedness = (double)ascendingPairs / (features.size - 1);
234         features.reversedness = (double)descendingPairs / (features.size -
235 1);
236
237         // Calculate uniqueness (use full dataset for accuracy)
238         unordered_set<int> uniqueElements(data.begin(), data.end());
239
240         features.uniqueCount = uniqueElements.size();
241         features.uniqueRatio = (double)uniqueElements.size() / features.
242 size;
243
244         // Classify dataset type
245         if (features.sortedness >= 0.80) features.type = "Nearly Sorted";
246         else if (features.reversedness >= 0.90) features.type = "Reversed";
247         else if (features.uniqueRatio < 0.40) features.type = "Few Unique";
248         else if (features.isLargeDataset) features.type = "Large Random";
249         else features.type = "Random";
250
251         return features;
252     }
253
254     // Predict best sorting algorithm based on dataset features
255     static AlgoType predictBestAlgorithm(const DatasetFeatures& features) {
256         // AI Decision Tree based on algorithm complexity theory
257
258         // Rule 1: Very small datasets (Size <= 50)
259         // Insertion Sort has low constant factor, faster than Quick/Merge
260         recursion overhead
261         if (features.size <= 50) {
262             return INSERTION_SORT;
263         }

```



```

261
262 // Rule 2: Large datasets (Size > 1000)
263 if (features.isLargeDataset) {
264     // Few unique values: Merge Sort is more stable than Quick Sort
265     if (features.uniqueRatio < 0.40) {
266         return MERGE_SORT;
267     }
268     return QUICK_SORT;
269 }
270
271 // Rule 3: Medium-sized datasets (50 < Size <= 1000)
272
273 // Case A: Nearly sorted
274 // Insertion Sort degrades to O(N) for nearly sorted data
275 if (features.sortedness >= 0.80) {
276     return INSERTION_SORT;
277 }
278
279 // Case B: Reversed
280 // Merge Sort is better for reversed data (stable O(N log N))
281 // Quick Sort with fixed pivot has O(N^2) worst case on reversed
data
282 if (features.reversedness >= 0.90) {
283     return MERGE_SORT;
284 }
285
286 // Case C: Few unique values
287 if (features.uniqueRatio < 0.40) {
288     return MERGE_SORT;
289 }
290
291 // Case D: Random data
292 return QUICK_SORT;
293 }
294
295 // Get algorithm name from type
296 static string getAlgoName(AlgoType type) {
297     switch (type) {
298         case BUBBLE_SORT: return "Bubble Sort";
299         case INSERTION_SORT: return "Insertion Sort";
300         case MERGE_SORT: return "Merge Sort";
301         case QUICK_SORT: return "Quick Sort";
302         default: return "Unknown";
303     }
304 }
305
306 // Run sorting algorithm and measure performance
307 static SortMetrics runSort(AlgoType type, vector<int> data) {
308     SortMetrics metrics;
309     metrics.algoName = getAlgoName(type);
310     metrics.comparisons = 0;
311
312     auto start = chrono::high_resolution_clock::now();
313
314     switch (type) {
315         case BUBBLE_SORT: bubbleSort(data, metrics.comparisons); break;
316         case INSERTION_SORT: insertionSort(data, metrics.comparisons);
break;

```

```

317         case MERGE_SORT: mergeSort(data, 0, data.size() - 1, metrics.
comparisons); break;
318         case QUICK_SORT: quickSort(data, 0, data.size() - 1, metrics.
comparisons); break;
319     }
320
321     auto end = chrono::high_resolution_clock::now();
322     chrono::duration<double, milli> duration = end - start;
323     metrics.executionTimeMs = duration.count();
324
325     return metrics;
326 }
327 };
328
329 // ===== Qt Visualization Interface =====
330
331 class SortingVisualizer : public QMainWindow {
332     Q_OBJECT
333
334 private:
335     // UI Components
336     QComboBox* datasetTypeCombo;
337     QSpinBox* dataSizeSpinBox;
338     QSpinBox* uniqueCountSpinBox;
339     QLabel* uniqueCountLabel;
340     QPushButton* generateBtn;
341     QPushButton* runBtn;
342     QTextEdit* dataPreviewText;
343     QTextEdit* analysisResultText;
344     QTableWidgetItem* resultsTable;
345     QLabel* statusLabel;
346
347     // Data
348     vector<int> currentDataset;
349
350 public:
351     SortingVisualizer(QWidget *parent = nullptr) : QMainWindow(parent) {
352         setWindowTitle("AI-Driven Sorting Algorithm Optimizer");
353         resize(900, 650);
354
355         QWidget* central = new QWidget(this);
356         setCentralWidget(central);
357         QVBoxLayout* mainLayout = new QVBoxLayout(central);
358
359         // Dataset Generation Area
360         QGroupBox* genGroup = new QGroupBox("Dataset Generation");
361         QHBoxLayout* genLayout = new QHBoxLayout(genGroup);
362
363         genLayout->addWidget(new QLabel("Type:"));
364         datasetTypeCombo = new QComboBox();
365         datasetTypeCombo->addItem({"Random", "Nearly Sorted", "Reversed",
"Few Unique", "Large Random"});
366         genLayout->addWidget(datasetTypeCombo);
367
368         genLayout->addWidget(new QLabel("Size:"));
369         dataSizeSpinBox = new QSpinBox();
370         dataSizeSpinBox->setRange(10, 100000);
371         dataSizeSpinBox->setValue(1000);

```

```

372     dataSizeSpinBox->setSingleStep(100);
373     genLayout->addWidget(dataSizeSpinBox);
374
375     uniqueCountLabel = new QLabel("Unique:");
376     uniqueCountSpinBox = new QSpinBox();
377     uniqueCountSpinBox->setRange(2, 50);
378     uniqueCountSpinBox->setValue(5);
379     uniqueCountSpinBox->setEnabled(false);
380     genLayout->addWidget(uniqueCountLabel);
381     genLayout->addWidget(uniqueCountSpinBox);
382
383     generateBtn = new QPushButton("Generate Dataset");
384     generateBtn->setStyleSheet("background-color: #4CAF50; color: white
; font-weight: bold; padding: 8px;");
385     genLayout->addWidget(generateBtn);
386
387     runBtn = new QPushButton("Run Analysis and Sort");
388     runBtn->setStyleSheet("background-color: #2196F3; color: white;
font-weight: bold; padding: 8px;");
389     runBtn->setEnabled(false);
390     genLayout->addWidget(runBtn);
391
392     genLayout->addStretch();
393     mainLayout->addWidget(genGroup);
394
395     // Data Preview Section
396     QGroupBox* previewGroup = new QGroupBox("Data Preview");
397     QVBoxLayout* previewLayout = new QVBoxLayout(previewGroup);
398     dataPreviewText = new QTextEdit();
399     dataPreviewText->setReadOnly(true);
400     dataPreviewText->setMinimumHeight(100);
401     dataPreviewText->setPlaceholderText("Click 'Generate Dataset' to
start...");
402     previewLayout->addWidget(dataPreviewText);
403     mainLayout->addWidget(previewGroup);
404
405     // AI Analysis Results Section
406     QGroupBox* analysisGroup = new QGroupBox("AI Analysis Results");
407     QVBoxLayout* analysisLayout = new QVBoxLayout(analysisGroup);
408     analysisResultText = new QTextEdit();
409     analysisResultText->setReadOnly(true);
410     analysisResultText->setMinimumHeight(120);
411     analysisLayout->addWidget(analysisResultText);
412     mainLayout->addWidget(analysisGroup);
413
414     // Performance Comparison Table
415     QGroupBox* resultsGroup = new QGroupBox("Sorting Performance
Comparison");
416     QVBoxLayout* resultsLayout = new QVBoxLayout(resultsGroup);
417     resultsTable = new QTableWidgetItem();
418     resultsTable->setColumnCount(3);
419     resultsTable->setHorizontalHeaderLabels({"Algorithm", "Comparisons
", "Time(ms)"});
420     resultsTable->horizontalHeader()->setSectionResizeMode(QHeaderView
::Stretch);
421     resultsTable->setEditTriggers(QAbstractItemView::NoEditTriggers);
422     resultsTable->setMinimumHeight(240);
423     resultsLayout->addWidget(resultsTable);

```

```

424     mainLayout->addWidget(resultsGroup);
425
426     // Status Bar
427     statusLabel = new QLabel("Ready");
428     mainLayout->addWidget(statusLabel);
429
430     // Connect Signals and Slots
431     connect(generateBtn, &QPushButton::clicked, this, &
SortingVisualizer::onGenerate);
432     connect(runBtn, &QPushButton::clicked, this, &SortingVisualizer::
onRun);
433     connect(datasetTypeCombo, QOverload<int>::of(&QComboBox::
currentIndexChanged), [this](int index) {
434         // Enable unique count spinbox only for "Few Unique" (index 3)
435         uniqueCountSpinBox->setEnabled(index == 3);
436         uniqueCountLabel->setEnabled(index == 3);
437     });
438 }
439
440 private slots:
441     void onGenerate() {
442         int size = dataSizeSpinBox->value();
443         int type = datasetTypeCombo->currentIndex();
444
445         statusLabel->setText("Generating...");
446
447         try {
448             // Generate dataset based on selected type
449             switch(type) {
450                 case 0: currentDataset = SortingEngine::
generateRandomDataset(size); break;
451                 case 1: currentDataset = SortingEngine::
generateNearlySorted(size); break;
452                 case 2: currentDataset = SortingEngine::generateReversed(
size); break;
453                 case 3: currentDataset = SortingEngine::generateFewUnique(
size, uniqueCountSpinBox->value()); break;
454                 case 4:
455                     // Large Random Dataset
456                     if (size < 10000) {
457                         QMessageBox::information(this, "Info", "Large
Random Dataset requires minimum size of 10000. Size adjusted to 10000.")
;
458                         size = 10000;
459                         dataSizeSpinBox->setValue(10000);
460                     }
461                     currentDataset = SortingEngine::generateRandomDataset(
size);
462                     break;
463             }
464
465             // Display Preview
466             ostringstream oss;
467             int preview = min(40, size);
468             oss << "Size: " << size << " | First " << preview << " elements
: [";
469             for (int i = 0; i < preview; i++) {
470                 oss << currentDataset[i];

```

```

471         if (i < preview - 1) oss << ", ";
472     }
473     if (size > preview) oss << ", ...";
474     oss << "]";
475     dataPreviewText->setText(QString::fromStdString(oss.str()));
476
477     runBtn->setEnabled(true);
478     statusLabel->setText("Dataset Generated Successfully");
479     analysisResultText->clear();
480     resultsTable->setRowCount(0);
481
482     } catch (const exception& e) {
483         QMessageBox::critical(this, "Error", e.what());
484         statusLabel->setText("Generation Failed");
485     }
486 }
487
488 void onRun() {
489     if (currentDataset.empty()) {
490         QMessageBox::warning(this, "Warning", "Please generate a
dataset first");
491         return;
492     }
493
494     statusLabel->setText("Analyzing...");
495     QApplication::processEvents();
496
497     // AI Analysis
498     DatasetFeatures features = SortingEngine::analyzeDataset(
currentDataset);
499     AlgoType predicted = SortingEngine::predictBestAlgorithm(features);
500
501     // Display analysis results
502     ostringstream oss;
503     oss << "[Dataset Features]\n";
504     oss << "Type: " << features.type << " | ";
505     oss << "Size: " << features.size << (features.isLargeDataset ? " (
Large)" : " (Small/Medium)") << "\n";
506     oss << "Sortedness: " << fixed << setprecision(1) << (features.
sortedness * 100) << "% | ";
507     oss << "Reversedness: " << (features.reversedness * 100) << "% | ";
508     oss << "Uniqueness: " << (features.uniqueRatio * 100) << "%\n\n";
509     oss << "[AI Prediction] Optimal Algorithm: " << SortingEngine::
getAlgoName(predicted);
510
511     analysisResultText->setText(QString::fromStdString(oss.str()));
512
513     statusLabel->setText("Sorting...");
514     QApplication::processEvents();
515
516     // Run Sorting Algorithms
517     vector<SortMetrics> results;
518     int size = currentDataset.size();
519
520     // Skip O(n^2) algorithms for large datasets
521     if (size <= 1000) {
522         results.push_back(SortingEngine::runSort(BUBBLE_SORT,
currentDataset));

```

```

523         results.push_back(SortingEngine::runSort(INSERTION_SORT,
currentDataset));
524     }
525     results.push_back(SortingEngine::runSort(MERGE_SORT, currentDataset
));
526     results.push_back(SortingEngine::runSort(QUICK_SORT, currentDataset
));
527
528     // Find the best performing algorithm
529     string actualBest;
530     double minTime = 1e9;
531     for (const auto& r : results) {
532         if (r.executionTimeMs < minTime) {
533             minTime = r.executionTimeMs;
534             actualBest = r.algoName;
535         }
536     }
537
538     // Display Results in Table
539     resultsTable->setRowCount(results.size());
540     for (size_t i = 0; i < results.size(); i++) {
541         QTableWidgetItem* nameItem = new QTableWidgetItem(QString::
fromStdString(results[i].algoName));
542         QTableWidgetItem* compItem = new QTableWidgetItem(QString::
number(results[i].comparisons));
543         QTableWidgetItem* timeItem = new QTableWidgetItem(QString::
number(results[i].executionTimeMs, 'f', 4));
544
545         // Highlight the best performing algorithm
546         if (results[i].algoName == actualBest) {
547             QBrush gold(QColor(255, 215, 0, 120));
548             nameItem->setBackground(gold);
549             compItem->setBackground(gold);
550             timeItem->setBackground(gold);
551         }
552
553         resultsTable->setItem(i, 0, nameItem);
554         resultsTable->setItem(i, 1, compItem);
555         resultsTable->setItem(i, 2, timeItem);
556     }
557
558     // Update status with prediction accuracy
559     string predictedName = SortingEngine::getAlgoName(predicted);
560     if (predictedName == actualBest) {
561         statusLabel->setText("Complete | AI Prediction Correct! Best: "
+ QString::fromStdString(actualBest));
562     } else {
563         statusLabel->setText("Complete | Predicted: " + QString::
fromStdString(predictedName) +
564             " -> Actual Best: " + QString::fromStdString
(actualBest));
565     }
566 }
567 };
568
569 // ===== Main Function =====
570
571 int main(int argc, char *argv[]) {

```

```
572     QApplication app(argc, argv);
573     SortingVisualizer window;
574     window.show();
575     return app.exec();
576 }
577
578 #include "Cui_Zeyu_DSC2409006_CST207_Project_Group_202509_GUI.moc"
```

Listing 2: Cui_Zeyu_DSC2409006_CST207_Project_Group_202509_GUI.cpp

APPENDIX 1

MARKING RUBRICS

Component Title	Group Work					Percentage (%)	
Criteria	Score and Descriptors					Weight (%)	Marks
	Excellent (5)	Good (4)	Average (3)	Need Improvement (2)	Poor (1)		
Dataset Generation (CLO 3)	All required datasets are implemented correctly	Minor missing datasets	Some datasets implemented	Many datasets missing	Dataset generation not implemented	10	
Sorting Algorithms (CLO 3)	All required algorithms implemented correctly and efficiently	Minor inefficiencies or small errors	Most algorithms implemented	Major issues or missing algorithms	Algorithms not implemented	20	
AI Module (CLO 3)	AI module correctly predicts best sorting algorithm	Minor issues in predictions or logic	AI module works partially	AI predictions mostly incorrect	AI module not implemented	20	
Performance Measurement (CLO 3)	All algorithms measured correctly with time & comparisons	Minor mistakes in measurements	Measurements mostly correct	Significant errors	No measurements	15	
Code Clarity & Documentation (CLO 3)	The code demonstrates excellent readability and clarity.	Minor readability issues	Some unclear code	Poorly documented	Code unreadable	10	
Report Quality (CLO 3)	Well-organized, complete report with clear analysis	Minor formatting or clarity issues	Adequate report with missing details	Poorly organized report	Report missing or unreadable	10	
						85	

Note to students: Please include the marking rubric when submitting your coursework.

Component Title	Individual Work					Percentage (%)	
Criteria	Score and Descriptors					Weight (%)	Marks
	Excellent (5)	Good (4)	Average (3)	Need Improvement (2)	Poor (1)		
Individual Contribution (CLO 3)	Clear, detailed description of individual work	Minor details missing	Contribution mentioned with little detail	Contribution vague or incomplete	Contribution not mentioned	10	
Presentation & Video (CLO 3)	Clear, well-paced, complete video	Minor pacing issues	Adequate video	Poor video clarity or incomplete	No video submitted	5	
						15	

Note to students: Please include the marking rubric when submitting your coursework.